A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures

BIN REN, The Ohio State University
TODD MYTKOWICZ, Microsoft Research
GAGAN AGRAWAL, The Ohio State University

Fine-grained data parallelism is increasingly common in the form of longer vectors integrated with mainstream processors (SSE, AVX) and various GPU architectures. This article develops support for exploiting such data parallelism for a class of nonnumeric, nongraphic applications, which perform computations while traversing many independent, irregular data structures. We address this problem by developing several novel techniques. First, for code generation, we develop an intermediate language for specifying such traversals, followed by a runtime scheduler that maps traversals to various SIMD units. Second, we observe that good data locality is crucial to sustained performance from SIMD architectures, whereas many applications that operate on irregular data structures (e.g., trees and graphs) have poor data locality. To address this challenge, we develop a set of data layout optimizations that improve spatial locality for applications that traverse many irregular data structures. Unlike prior data layout optimizations, our approach incorporates a notion of both interthread and intrathread spatial reuse into data layout. Finally, we enable performance portability (i.e., the ability to automatically optimize applications for different architectures) by accurately modeling the impact of inter- and intrathread locality on program performance. As a consequence, our model can predict which data layout optimization to use on a wide variety of SIMD architectures.

To demonstrate the efficacy of our approach and optimizations, we first show how they enable up to a 12X speedup on one SIMD architecture for a set of real-world applications. To demonstrate that our approach enables performance portability, we show how our model predicts the optimal layout for applications across a diverse set of three real-world SIMD architectures, which offers as much as 45% speedup over a suboptimal solution.

Categories and Subject Descriptors: D.13 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Irregular data structure, fine-grained parallelism, SIMD

ACM Reference Format:
Bin Ren, Gagan Agrawal, and Todd Mytkowicz. 2014. A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures. ACM Trans. Architec. Code Optim. 11, 2, Article 16 (June 2014), 31 pages.
DOI: http://dx.doi.org/10.1145/2632215

This work was partially supported by NSF award CCF-1319420. Extension of a conference paper: An earlier version of this article appeared in the 2013 International Symposium on Code Generation and Optimization (CGO '13) [Ren et al. 2013]. The new material in this article includes the following: (1) we extend the interpretation framework from being solely for the Intel SSE architecture to include multiple GPUs architectures; (2) we carefully study the impact of interthread and intrathread spatial data reuse on performance on various SIMD architectures; (3) to enable performance portability, we have designed an analytic model for choosing the layout, and (4) we evaluate our methods on all three architectures and show good performance.
Authors' addresses: Bin Ren and Gagan Agrawal, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, 43210; Todd Mytkowicz, Microsoft Research, One Microsoft Way, Redmond, WA, 98052; emails: {ren, agrawal}@cse.ohio-state.edu; toddm@microsoft.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1544-3566/2014/06-ART16 $15.00
DOI: http://dx.doi.org/10.1145/2632215

1. INTRODUCTION

Fine-grained data parallelism is becoming increasingly prevalent in mainstream processors, such as x86 and ARM, as the length of vector instructions is increasing.[1] The most common fine-grained data-parallel hardware, the Steaming SIMD Extensions (SSE), has been part of x86 since 1999 and is widely used in graphics [Ide et al. 2000], image, video, and signal processing [Franchetti and Puschel 2002], as well as scientific and engineering applications [Garca et al. 2003; Rognes and Seeberg 2000]. As such fine-grained data parallelism becomes a ubiquitous processor feature with increasing performance, it is desirable to exploit this feature for irregular computations as well.

However, programs that rely on irregular, pointer-based data structures benefit little from SIMD execution because of the mismatch between the strict, lockstep behavior of SIMD parallelism and the dynamic, data-driven behavior of programs that manipulate irregular data structures. This article starts to bridge this gap by demonstrating an approach to speeding a class of applications that involve independent traversals of many instances of a pointer-based data structure. Examples of the class of applications we target include prediction using a collection of decision trees [Breiman 2001], matching with regular expressions [Vasiliadis et al. 2009], parsing XML documents [Nicola and John 2003], and frequent pattern mining [Han et al. 2007], including finding common subgraphs in a set of graphs [Yan and Han 2002]. These applications arise in domains as diverse as machine learning, compilation, intrusion detection, Web services, databases, and data mining.

These applications traverse independent data structures for two reasons. First, applications can manipulate a large number of logically independent data structures. For example, the forest of decision trees produced by a machine learner or the set of alternative patterns used in an intrusion detection system such as Snort [Roesch 1999] represent independent computations that can proceed in parallel. Second, applications can traverse a single irregular data structure, but do so with many independent inputs. The approach in this work handles both cases. Although applications of this type can easily be parallelized across multiple cores, SIMD within each core can provide a multiplicative improvement in performance.

Effective parallelization of independent traversals of irregular data structures on a SIMD unit requires addressing multiple challenges. One such challenge is related to the uneven amount of work that each SIMD unit might have to perform while traversing different structures. Another challenge is that these applications involve branch operations, which cannot be parallelized on SIMD units. Memory latency while traversing pointer-based data structures is another issue.

This article develops techniques to address these problems. Moreover, we offer a solution to programmers interested in developing SIMD parallelized implementations of these applications, by developing an intermediate language and a runtime scheduler. The intermediate language exposes several types of operations, which can be used to specify the traversal involved in the application. We implement several optimizations in our runtime scheduler, including a stream compaction method, several layouts that reduce memory latency (while also allowing branch operations to be replaced by arithmetic operations).

In addition to code generation for SIMD, a key architectural feature that programmers need to optimize is the memory hierarchy. In sequential applications, a programmer need only consider locality from the viewpoint of a single thread (e.g., *intrathread* locality). However, for modern SSE-based processors, multicore, or SIMD architectures, the memory hierarchy is shared by multiple threads, so locality, shared among different

[1]For example, the Intel Sandybridge processor doubled its vector length to 256 bits.

threads (*interthread* locality), can dominate an application's performance [Meng et al. 2010; Unkule et al. 2012].

Clearly, it would be desirable for a code generator to automatically optimize an application's locality for a given architecture, thereby removing the burden of performance portability from the programmer. This work demonstrates how to enable performance portability across a wide variety of SIMD architectures. Overall, we present a domain-specific optimization framework that targets a variety of architectures that use SIMD parallelism. This includes an x86 system with SSE instructions, and two generations of NVIDIA GPUs. Each of these architectures differs significantly in their memory hierarchies and width of SIMD units. Our optimization framework includes a family of data layout optimizations that help achieve interthread spatial locality, intrathread spatial locality, or a combination of the two.

We find that the relative performance among different layouts depends on an application's characteristics (e.g., whether all trees are processed with equal likelihood, or if probability of taking all branches is the same or not), as well as architectural characteristics (e.g., features of memory hierarchy and degree of SIMD parallelism). Thus, we develop a detailed performance model that can help select an appropriate layout for a given application/architecture combination. The key insight of our model—and why we are able to accurately model real-world applications running on diverse machines—is that for the applications in our restricted domain, the number of L2 cache misses turns out to be an effective predictor of performance.

Overall, the contributions of this work are:

—Identification of an opportunity to exploit fine-grain data parallelism in important, latency critical algorithms widely used in production software.
—An approach that exploits fine-grained parallelism when traversing pointer-based data structures, with a specific emphasis on trees and graphs.
—Three novel data layout optimizations that are designed to extract intra- and/or interthread locality from applications that traverse a large number of irregular data structures on SIMD hardware.
—An analytic model that removes the burden of performance portability from the programmer by accurately modeling which of our data layout optimizations to use on a particular architecture.
—An illustration of the practicality of our approach by demonstrating significant single-core speedups of three applications that use irregular data structures on one SIMD architecture with 9X, 12X, and 5X speedup, respectively. Similarly, we demonstrate the efficacy of our data layout optimizations and cache analytic model by showing significant speedups of two real-world applications with different input datasets on three diverse SIMD architectures. Our experimental results show that our model is able to capture the affect factors of performance from both the architecture side and application side, which provides optimized configurations for the users' program and offers as much as around 45% speedup compared to the suboptimal solution.

2. ANATOMY OF THREE IRREGULAR PROGRAMS

This section introduces three common algorithms that manipulate a large number of irregular data structures—random forest, B+ trees, and regular expression matching. All of these algorithms have a significant amount of task-level parallelism, as both traverse a large number of independent, irregular trees or graphs.

2.1. Random Forest

Random forests are a data mining technique used to classify an input—or a set of features—into a fixed number of categories [Breiman 2001]. A random forest is a

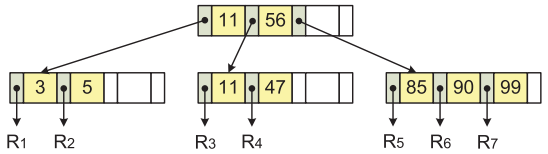
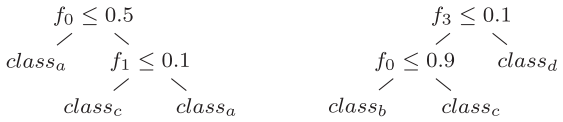


Fig. 1. An example of B+ tree structure.

collection of binary decision trees. To classify an input, each tree is traversed, comparing features of the input against threshold values, and producing a result as its categorical membership for that input. The input is then classified according to the category with the most votes. The intuition behind a random forest is that if each tree's vote is slightly better than a random guess, a large number of trees will be able to—on average—correctly classify an input.

To be more concrete, consider a random forest made of two simple trees, shown next. Each tree can classify one input, made of three features (f_0, f_1, and f_3), into one of four classes ($class_a$, $class_b$, $class_c$, and $class_d$):



Each node in the two trees performs one of two actions. If a node is an internal node, it compares one feature of the input against a constant threshold and branches accordingly to a left or right child depending on the result of the comparison. If the node is a leaf node, it simply stores the class label for that tree into a global counter.

Random forests usually contain many trees, and each tree has far more nodes than in this simple example. Each traversal is a unique task, independent of other traversals, and can execute in parallel. For example, given the preceding example, we could evaluate each tree independently on a different core of a multicore machine.

2.2. B+ Tree

B+ tree is a popular indexing structure used in databases (e.g., for indexing) and data servers (e.g., for file systems) [Comer 1979]. The basic B+ tree operations are as follows: for an internal node, a comparison operation is performed between the input value and the key values, and for a leaf node, an output operation is performed to generate the searched record.

Consider the simple B+ tree example in Figure 1. For an input query with an integer number as its key, we compare its key against the key values in an internal node and perform a branch operation. This process is repeated until we arrive at the leaf node, where the information about the searched record is output. A single B+ tree can be represented as a hashing function in conjunction to a large number of independent subtrees.[2] The hashing function represents the logic of the first n levels of the tree—that is, it is a mechanism by which we can quickly reference any one of the 2^n subtrees.

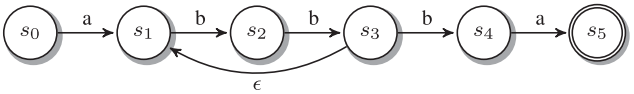
In a database or data server where a B+ tree is deployed, it is common to have a heavy query workload, with several independent inputs at any given time, which makes the application suitable for exploiting parallelism. After the hashing function is applied to each input, there can be independent queries on different subtrees. These

[2]http://publib.boulder.ibm.com/infocenter/idshelp/v117/index.jsp?topic=%2Fcom.ibm.perf.doc%2Fids_prf_763.htm.

are analogous to independent traversal of different weak classifiers for a single input in random forests, with one key difference—not all subtrees might be concurrently processed at any given time. We refer to this traversal pattern as sparse buckets accesses. Overall, this application is distinct from random forest because parallelism is arising from processing of different inputs, and yet, the issues in efficient SIMD execution are almost the same, as we elaborate next.

2.3. Regular Expression Matching

Regular expressions are a common way to match patterns against large bodies of text or binary data. In this article, we use a nondeterministic finite automaton (NFA) to simulate a regular expression, similar to Thompson's original regular expression compiler [Thompson 1968]. To simulate an NFA, we walk—or traverse—a graph, moving node to node in the graph depending on the type of action required at a node. Consider the NFA, or graph, for a simple regular expression $a(bb) + ba$:



To evaluate this regular expression, we traverse the NFA, starting at node s_0. If the traversal ever reaches node s_5, the regular expression matches the input string. To traverse from one node to another, we compare the input character to the character on that node's outgoing edge. If the input character matches the edge character, we follow that edge to the next node and move forward one character in the input. If it is not a match, then we stop the traversal, as the input string is not a match. If the edge character is ϵ, we traverse the edge without advancing the input.

Traversing the graph has one added complication—nodes may have two edges. When walking the graph, if our traversal comes to a node with two outgoing edges, then we must follow both edges. That is, we try both paths at the same time, reading the input only once. The result of the traversal is the union of both traversals. For example, suppose that we are in node s_3, then due to the edge labeled ϵ, we start our multinode traversal in node s_1 and s_3 simultaneously. If the next input is a "b", both traversals advance. If the next input is an "a", the set of active traversals narrows down to a single traversal.

2.4. Challenges to Efficient Execution

These applications—and many like them that traverse irregular data structures—face two significant challenges that limit efficient execution.

Fine-grained parallelism. The programs we described earlier clearly have a large amount of *coarse-grained* parallelism. For example, simultaneous traversals of a large number of trees can be easily divided between several cores of a multicore machine. However, these programs also have a significant amount of *fine-grained* parallelism—for example, every internal node executes a compare. Exploiting fine-grained parallelism (e.g., via SIMD in SSE or GPUs) is complementary to coarse-grained parallelism and can speed each core's processing of its portion of the set of nodes. However, mapping this fine-grained parallelism to SIMD hardware is nontrivial.

Memory locality. Programs that traverse a large number of irregular data structures likely have no temporal reuse and poor spatial locality. For example, if nodes of a tree are allocated on the heap (i.e., using memory allocators like `malloc`), there is no assurance as to where a node's children are allocated with respect to the parent. Even

if a programmer is smart and linearizes the irregular data structure into a dense block of memory, high spatial reuse may not occur. Assuming a balanced tree, there are 2^n nodes at the level n of any given tree, among which only one node will be accessed during any traversal. Thus, once we reach a level where the nodes at the level occupy more than one cache line, we will not see any spatial reuse. As a consequence, it is common for a program that uses irregular data structures to be stalled on memory, which in turn negates any possible benefits from parallelism.

This article focuses on these two important challenges: SIMD code generation and execution, and locality. To address the first challenge, we have developed an intermediate language for specifying the irregular traversals, followed by a runtime scheduler that maps traversals to SIMD units. To address the second challenge, we have designed a set of data layouts, with enabling interthread and/or intrathread data locality, and an analytical model that allows us to automatically choose the right layout for a given architecture.

3. SIMD TRAVERSAL OF FINE-GRAINED TASKS

We now focus on the problem of executing irregular applications, like regular expressions or decision/B+ trees, on SIMD hardware. Our work targets both SSE-like SIMD hardware, as well as a variant seen in modern GPUs, which is also referred to as the Single Instruction Multiple Threads (SIMT) architecture. SIMT is a more relaxed architecture in the sense that it can manage divergence between threads at the hardware level. Therefore, our approach is described using the SSE architecture. The work has been extended to GPU architectures without many modifications.

Returning to our target class of applications, when these programs visit an internal node, they optionally perform an operation on it, and might also include evaluating an expression to select the next node to visit. On visiting a leaf node, a different operation is performed (e.g., storing the result), as there is no new node to be visited. We denote the operation(s) performed at a leaf node n as $\mathcal{W}(n)$ and the computations at the nonleaf node as $\mathcal{T}(n)$.

As a concrete example, consider the decision trees in Section 2.1 and a typical sequential single decision tree traversal. Each node of this tree is either an internal node or a leaf node. When we traverse to an internal node n, we compare feature values and branch to the left or right child, depending on the result. When we reach a leaf node n, we update the ranking and terminate the traversal. Thus, for this application, the former is $\mathcal{T}(n)$ and the latter is $\mathcal{W}(n)$.

3.1. High-Level Approach to SIMD Execution

Now, suppose that we want to traverse a set of trees on SIMD hardware. We can first execute $\mathcal{T}(n)$ for all root nodes n, which gives us a set of successor nodes ns. Next, assuming that all tree traversals are of the same length, we can evaluate $\mathcal{T}(n)$ for all n in ns, and so on, until we reach a leaf node. At a leaf node n, we evaluate $\mathcal{W}(n)$ for all trees.

However, in practice, and unlike a typical array-based computation, the different traversals likely have different lengths. Therefore, at a certain level, a mixture of $\mathcal{T}$ and $\mathcal{W}$ computations will be needed. Thus, we must execute each operation type for each stage and mask the results of operations whose types are not represented by the current operation. What we are doing is essentially emulating MIMD with SIMD, a topic that has been studied in the past [Hanxleden and Kennedy 1992; Dietz and Cohen 1992; Blelloch et al. 1994; Hardwick 1996]. However, none of this work has considered pointer-based traversals.

The second problem is that SIMD execution requires that addressing children is branchless; otherwise, we are unable to parallelize the $\mathcal{T}(n)$ expressions on SSE-like

ALGORITHM 1: Interpreter (*byte_codes*, *task_queue*)

```
1: result = 0
2: ▷ Initialize task_queue by adding root-level tasks
3: task_queue = Initialize(roots)
4: for n ∈ task_queue by SIMD-Width do
5:    ▷ Process traversal operations in SIMD
6:    ns = T(bytecodes, n)
7:    ▷ Identify finished traversals
8:    isLeaf = findIsLeaf(ns)
9:    ▷ Strip out finished traversals
10:   ns = streamcompact(ns,isLeaf)
11:   task_queue.push_back(ns)
12:   ▷ Process W operations according to isLeaf
13:   result = W(bytecodes, isLeaf)
14: end for
15: return result
```

architectures. To address this problem, we design a layout generation process to organize data structure elements in the memory in a systematic way. We can provide a uniform interface so that the details of the memory layout are transparent, but we must be able to address left and right children of a node with arithmetic operations. Specifically, suppose that a node has zero, one, or two children. If a node has two children, we store the left child contiguous to the right in memory. This organization works well for SIMD addressing, as we can use a simple arithmetic operation to address the left and right children of a node. We require that all $\mathcal{T}(n)$ expressions return a 0 to branch to the left child and a -1 to branch right. Thus, for a given node n, if the *ns* field stores the location of the *left* child, the next node to visit is $ns - \mathcal{T}(n)$. In effect, this turns the addressing of children from a control dependence into a data dependence.

Formalizing this, we can put our approach together as an general method (Algorithm 1). Besides the solutions to the two key problems listed earlier, there are a couple of additional issues that we addressed in this algorithm. In some applications, it is necessary to dynamically fork, or start, new fine-grained tasks at a particular step. This operation needs to be (i) efficient and (ii) parallelizable in SIMD. For example, in our regular expression engine, we create a new fine-grained task whenever we traverse an ϵ node in a NFA. To fork a task, we introduce a fork instruction that starts a child fine-grained task at the location of its left child and continues the parent task at the location of the right child.

Further, because not all tasks finish at the same time, we need to remove tasks from processing when they reach a leaf node. Like stated previously, removing tasks must be efficient and not require complicated control flow. To efficiently remove tasks from processing, we use a data parallel technique called *stream compaction* [Chatterjee et al. 1990]. We discuss details of this optimization in Section 3.3.

3.2. A General Solution for Multiple Applications

SIMD parallelization of each individual application following the methodology described earlier can be extremely hard. The programmers need to pay attention to a number of details and can easily write unoptimized and/or even incorrect code. To help development of applications, we have developed an intermediate language and a runtime scheduler or interpreter.

Our solution can be viewed as a virtual machine, where instructions from an intermediate language, or bytecodes, are executed on SIMD units. The bytecodes that we currently support are listed in Table I. Each bytecode is one of the two types: $\mathcal{T}(n)$

Table I. Bytecodes Supported by Our Interpreter and Their Semantics

Bytecode	Arg	Type	Description
match	None	$\mathcal{W}(n)$	Found a match; record position; terminate task.
nomatch	None	$\mathcal{W}(n)$	Found no match; terminate task.
store	`float`	$\mathcal{W}(n)$	Store the arg part of current bytecode to results.
cmp	`char/float`	$\mathcal{T}(n)$	Advance PC according to the comparison result.
dot	None	$\mathcal{T}(n)$	Advance PC by 1 on any input; if input is null, set PC to nomatch.
jmp	`char`	$\mathcal{T}(n)$	Set PC to argument.
fork	`char`	$\mathcal{T}(n)$	Fork a thread: advance parent PC by 1 and set child PC to arg.

Table II. Random Forest Using the Bytecodes

Nodes Type	Bytecode Sequence
Internal	cmp a;
Leaf	store

Table III. NFA Regex Using the Bytecode

Regular Expression	Bytecode Sequence
$\mathcal{C}(a)$	cmp a;
$\mathcal{C}(.)$	dot;
e_1e_2	$\mathcal{C}(e_1)$; $\mathcal{C}(e_2)$;
$e_1\|e_2$	fork L2; $\mathcal{C}(e_1)$; jmp L3; L2: $\mathcal{C}(e_2)$; L3: . . . ;
$e?$	fork L2; $\mathcal{C}(e)$; L2: . . . ;
$e*$	L1: fork L2; $\mathcal{C}(e)$; jmp L1; L2: . . . ;
$e+$	L1: $\mathcal{C}(e)$; fork L1;

and $\mathcal{W}(n)$, representing nonleaf and leaf operations, respectively. Any application that can be implemented using this operation can be mapped to SIMD hardware by our interpreter.

To show the generality of our approach, we have implemented all of the decision forest, B+ tree, and regular expression matching applications using our interpreter. Table II shows the translation from a tree structure to our bytecodes from a subset of bytecodes that we listed in Table I. For SIMD execution for NFA regular expressions, the specific method that we use is along the lines of Cox's NFA engine [Cox 2007], which in turn is based on Thompson's work [Thompson 1968]. This approach has an asymptotic complexity of $O(nm)$, where n is the number of fine-grained tasks and m is the size of the input string. This is far better than a naive NFA interpreter, which can at worst run in $O(n^2)$. Table III shows how the implementation handles different cases, using the bytecode from Table I.

Now, returning to how our interpreter works, we summarize the sequential and SSE implementations of our virtual machine in Algorithms 2 and 3, respectively. Algorithm 2 interprets the bytecodes of all trees/graphs level by level (LL) sequentially. In each level, it fetches bytecodes indexed by the task queue and processes either a $\mathcal{T}(n)$ operation or a $\mathcal{W}(n)$ operation for each tree/graph according to the type of the bytecode. Considering a more general situation that different portions of input may be required for different bytecodes dynamically, and the input pointer may be advanced as line 6 of Algorithm 2, such as the regular expression application, we maintain two task queues (lists)—that is, *clist* and *nlist*—in which *clist* is to handle the current portion of input, and *nlist* is to handle the next portion. Especially for applications like random forest, the required input index is predecided by bytecodes, and we do not need to move the input pointer, so *clist* and *nlist* can be simply merged as one task queue. After each iteration, we update either *clist* or *nlist* according to the bytecodes, especially $\mathcal{T}(n)$ operations generate either *one* or *two* (task expansion) new tasks, and $\mathcal{W}(n)$ operations generate *zero* tasks.

ALGORITHM 2: SeqInterpreter (*byte_codes*, *input*)

```
1: ▷ Initialize the result accumulator and task queue
2: result = 0 {*Accumulator for results*}
3: vector<> clist = Initialize(roots) {*Current list of PCs*}
4: while input! = NULL do
5:    ▷ If necessary, advance the input pointer
6:    input += AppShift
7:    vector<> nlist = Initialize(NULL) {*Next list of PCs*}
8:    while (!clist.empty()) do
9:       ▷ Get the bytecode indexed by clist
10:      pc = clist.pop_back() {*Pop a PC to execute*}
11:      op = byte_codes[pc]
12:      ▷ Process T(n) and W(n) operations
13:      if op.type == Bytecode :: cmp then
14:         nextPC = cmp(input, op.arg)
15:         nlist.push_back(nextPC)
16:      else if op.type == Bytecode :: dot then
17:         nlist.push_back(pc + 1)
18:      else if op.type == Bytecode :: jmp then
19:         clist.push_back(op.arg)
20:      else if op.type == Bytecode :: fork then
21:         clist.push_back(pc + 1)
22:         clist.push_back(op.arg)
23:      else if op.type == Bytecode :: match then
24:         result += 1
25:      else if op.type == Bytecode :: nomatch then
26:         result += 0
27:      else if op.type == Bytecode :: store then
28:         result += op.arg
29:         ▷ Jump to return statement
30:      end if
31:   end while
32:   swap(clist, nlist)
33: end while
34: return result
```

The SIMD interpreter described in Algorithm 3 is a SIMD parallel version of Algorithm 2, and a more detailed implementation of the overall method introduced in Algorithm 1. The basic logic of the SIMD execution part (line 14 to line 40) is as follows. We fetch multiple bytecodes indexed by the task queue elements according to the width of SIMD lanes and then load identical parts of multiple bytecodes into the same SIMD register, such as the *highBits* part identifying the type of bytecodes and the *args* part storing the address of next PC or output value. We next calculate various flags from types of bytecodes according to the highest bits to be able to mask invalid results. Finally, we process both $\mathcal{T}(n)$ and $\mathcal{W}(n)$ operations for all SIMD tasks and strip out the invalid results by the bytecode type flags calculated before. In the last stage, a stream compaction operation is used to remove the finished tasks, and thus to compact the task queues.

3.3. Light-Weight Stream Compaction for SSE Architecture

When different SIMD units are processing paths of different length, stream compaction is an important optimization to ensure SIMD efficiency. The basic idea is as follows. Suppose that we start off by needing to process 16 tree traversals concurrently. If the SIMD width is 4, processing the root level requires four iterations. Following that,

ALGORITHM 3: SIMDInterpreter (*byte_codes*, *input*)

```
1: ▷ Initialize the result accumulator and task queue
2: __m128 results = _mm_setzero_ps()
3: results_index = 0
4: clist[] = Initialize(roots)
5: clist_index = Initialize(roots)
6: while (input != NULL) do
7:    input += AppShift
8:    nlist[] = Initialize(NULL)
9:    nlist_index = 0
10:   while (clist_index > 0) do
11:      ▷ Copy clist for task creations
12:      tmplist = Initialize(clist)
13:      tmplist_index = clist
14:      for (i = 0; i < clist_length; i += SIMDWidth) do
15:         ▷ Get bytecodes indexed by clist in parallel
16:         __m128i PCIndexes = SIMDLoadPCIndexes(clist, i)
17:         __m128i ops = SIMDLoadCodes(byte_codes, PCIndexes)
18:         ▷ Get different parts of bytecodes parallel
19:         __m128i args = SIMDLoadArgs(ops) {*Args part*}
20:         __m128i highBits = SIMDLoadHiBits(ops){*High bits*}
21:         ▷ Decide types of ops in current SIMD lane by High bits
22:         __m128i isDot = SIMDcmp(ops, _mm_set1_epix(1))
23:         __m128i isJmp = SIMDcmp(highBits, jmpFlag)
24:         __m128i isFork = SIMDcmp(highBits, forkFlag)
25:         __m128i isMatch = SIMDcmp(ops, _mm_setzero_si128())
26:         __m128i isStore = SIMDcmp(highBits, storeFlag)
27:         ▷ Process T(n) operations and prepare new task queues
28:         ▷ 1. Execute the compare operation
29:         __m128i cmpResults = SIMDcmp(input, args)
30:         ▷ 2. Get addresses of nextPCs by types of operations
31:         __m128i address = SIMDcmp(highBits, ops)
32:         ▷ 3. Mask out the invalid nextPCs by isFork and isJmp
33:         __m128i nextAddress = SIMDcmp(isFork, isJmp, address)
34:         ▷ 4. Strip out finished nlist tasks, store rests to the proper position, advance nlist_index
35:         nlist_index += streamCompaction(cmpResults, isDot, nlist, nlist_index)
36:         ▷ 5. Similar operation on tmplist
37:         tmplist_index += streamCompaction(isJmp, isFork, nextAddress, tmplist, tmplist_
            index)
38:         ▷ Process W(n) operations in parallel
39:         results_index += streamCompaction(isMatch/isStore, results, results_index)
40:      end for
41:      swap(tmplist, clist)
42:      clist_index = tmplist_index
43:   end while
44:   swap(clist, nlist)
45:   clist_index = nlist_index
46: end while
47: return results
```

suppose that the list of nodes to be processed next is stored in an array of length 16, with a value of 0 denoting that the traversal is over. As an example, assuming that 5 traversals have been completed, whereas the other 11 traversals are still active, a simple execution mechanism may require four iterations to process these 11 traversals, as four consecutive values from the array are scheduled in one iteration. A more advanced

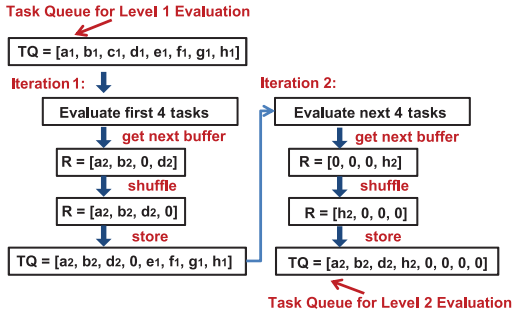


Fig. 2. An example of stream compaction for SIMD efficiency.

strategy might be to compact the nonzeroes in the array and use only three iterations to process the 11 nonzero entries.

We now explain our stream compaction method with the help of an example, shown in Figure 2. In this example, at first we have eight tasks in the initial task queue, and the SIMD lane width allows processing four tasks concurrently. Tasks c_1, e_1, f_1, and g_1 are leaf nodes, and after the first iteration of evaluation, we get one 0 in the corresponding position of c_1. Without stream compaction, a *bubble* task left in the SIMD lane undermines the utilization of parallelism. So, we utilize a *shuffle* operation to move the completed *bubble* tasks to the end of the SIMD lane, store the reordered tasks into the beginning position of the new task queue (*store_position* initialized as 0), and change *store_position* to 3. A similar operation is applied for the second iteration of evaluation, and the new generated tasks are stored into *store_position* (3), and the new *store_position* is increased to 4. If our application does not require task creation, we may use only one task queue to hold both the old and new tasks, since the number of new generated tasks is always smaller than or equal to the old ones, and it is impossible for the new generated tasks to overlap the unhandled old ones. However, for an application that involves task creation, we need to use two task queues to hold old tasks and new ones, respectively, and we swap them at the end of the evaluation of the same level of tasks.

Stream compaction can clearly improve the performance of our method by reducing the number of SIMD evaluation iterations of deeper levels with finished *bubble* tasks. For example, in Figure 2, without stream compaction, two iterations are required for level 2 evaluation, whereas with it, only one iteration is needed. Moreover, an interesting aspect of our implementation is that we can remove tasks from processing without complicated control flow (line 10 in Algorithm 1). Thus, we maintain parallel efficiency with only a small scheduling overhead.

3.4. Discussion: Extension to Other Architectures

Although our preceding description targeted the SSE architecture, the methods have been implemented on the GPU (SIMT) architectures as well. NVIDIA GPU architecture consists of multiple streaming multiprocessors (SMs), and each SM contains multiple streaming processors (SPs). SIMT parallelism is available within each SM—that is, all SPs must execute the same instruction in any given cycle. For example, Tesla C2050 (the Fermi architecture) has 14 SMs and 32 SPs per SM. Tasks are executed as warps, and each warp comprises threads running on the same SM. Thus, effectively, the SIMD width is 32.

To map our target class of applications onto a GPU, first we apply tiling to partition the bytecodes of irregular structures into multiple bags, where each bag contains

bytecodes of 32 trees or graphs. For each input, such as a feature vector for the random forest application, each bag of bytecodes is treated as a warp and is evaluated by multiple threads with the same SM. The basic idea for interpreting each warp is as described in Algorithm 1 (i.e., starting from all root nodes n), we need to execute both $\mathcal{T}(n)$ and $\mathcal{W}(n)$ operations for each level of the irregular structure to find the next level, store the valid results, and strip out the invalid results. During the interpretation, we use Algorithm 3 to handle the control flow dependency—that is, performing computations for all threads and masking out the invalid results. Alternatively, because SIMT is a more relaxed model than SIMD (i.e., branches can be handled by the hardware unlike in SSE-based processors), we can leverage the hardware support from GPU itself to handle the branches. These two approaches turn out to be equivalent to each other with the same overall performance in our test, and we use the latter in our implementation and experiments due to its simplicity. At the same time, multiple warps can also run on different SMs of a GPU concurrently, giving us a two-level or hierarchical parallelism.

There are two other implementation differences between our CPU and GPU implementations that we want to emphasize. One difference is that the GPU has its own memory hierarchy, so we need to transfer the bytecodes to the GPU before interpreting any inputs. The applications we evaluate in this article all involve evaluating thousands of inputs on a stable set of bytecodes, so we can amortize the cost of transferring the bytecodes to the GPU across many executions. Another significant difference is with respect to the stream compaction approach. Particularly, we do not have an efficient *shuffle* operation on the GPU architectures, since they have much wider SIMD lanes (32-way), as compared to 4 or 8-way lanes on SSE. Thus, we considered two alternatives. The first was the traditional *Prefix Sum*-based [Harris et al. 2007] stream compaction, and the second involved statically, at compile time, reordering the position of irregular structures according to the average traversal path lengths, and grouping the ones with similar access levels together. The first strategy introduced relatively large overheads, and thus we use the latter in our experiments. After reordering, 32 irregular structures with a similar access levels are grouped as one bag and assigned to one SM.

4. INTELLIGENT DATA LAYOUTS

A naive implementation of these applications, as well as other pointer-based applications, can be easily limited by the latency of the memory subsystem. For example, if every node of a tree or a graph is allocated using a library like `malloc`, there will likely be no spatial reuse, in addition to the fact that there is no temporal reuse in the application. Thus, improving memory efficiency is critical toward obtaining benefits from SIMD parallelize.

In this section, we present a number of layouts that not only linearize pointer-based irregular data structures into dense arrays but also improve spatial locality. The goals here include co-locating nodes (in the same cache line) that are likely to be accessed concurrently by different threads (interthread locality) and/or co-locating threads that are accessed in successive steps by the same thread (intrathread spatial locality).

First, we note that linearization also compresses a data structure. For example, a simple depth-first (DF) or breadth-first (BF) layout linearizes nodes of a tree based on the order they are visited in a DF or BF traversal (e.g., *DF* or *BF* in Figure 3(a)). By linearizing in a DF or BF layout, we can reduce the size of the irregular data structures; instead of two pointers, we only require a single pointer for each node in both linearized arrays.

However, both DF and BF linearize each tree independently of other trees and cannot improve interthread locality. While discussing various layouts, we use the notion of loading distance to measure interthread data locality. Specifically, the loading distance

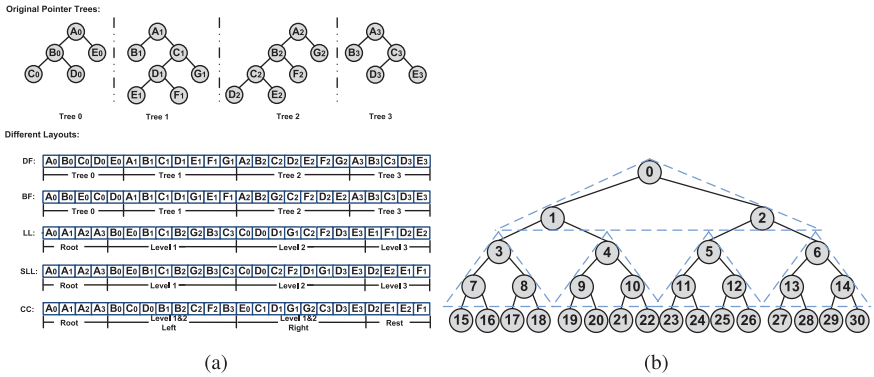


Fig. 3. (a) Memory layout with different schemes. (b) Cache-conscious (CC) layout for a single tree.

is the average distance between memory locations accessed at the same time, by consecutive threads. Clearly, as the loading distance increases, interthread data locality decreases. In DF or BF, the average loading distance between any two neighboring lanes is the number of nodes in the first of the two trees.

4.1. Improving Interthread Locality

To improve interthread locality, we introduce a layout that interleaves nodes across trees (shown as *LL* in Figure 3(a)). To be concrete, we first lay out the root nodes—the first tree's root node is followed by the second tree's root node, followed by the third, and so on. Then, we move on the next level. We co-locate a node's left and right children next to each other in memory. This allows us to use a single pointer to reference both children nodes, thus reducing the size to store the set of irregular data structures.

Interthread locality improvements through the use of this method is easy to see. For example, at the root level, nodes have large spatial locality; when we access the first tree's root node, we pull in the second tree's root node. If we have a complete and balanced binary tree, the loading distance is 2^l, where l corresponds to the depth of the tree. If the depth of each tree is k, then the loading distance varies between 1 and 2^k. In comparison, the loading distance for the BF and DF layouts are nearly 2^{k+1}. Thus, on the average, the loading distance is reduced, although more so for the initial levels of the tree than the lower levels.

Although the preceding LL scheme has several advantages, it does not utilize any possible bias in the traversal pattern. In many applications, there can be a greater likelihood of visiting one child above the other, and if this bias is known in advance, the more likely child can always be made the left child.

The next layout introduced is referred to as sorted level by level (SLL) and exploits such a bias to decrease the loading distance by a factor of up to two. An example of this layout is shown as the array *SLL* in Figure 3(a). In this layout, we divide the nodes of each tree at each level as *left* children and *right* children sets starting from the third level, in which all sibling nodes are still put together to support branchless addressing. We allocate the nodes in the left children set together followed by right ones for each level. For example, in the third level of Figure 3(a), we store the nodes in the left children set, $((C_0, D_0), (C_2, F_2))$, at first, followed by $((D_1, G_1), (D_3, E_3))$. When a tree is biased, we realize a reduced loading distance for each level of the tree.

4.2. Improving Intrathread Locality

Improving cache performance of pointer-traversing applications has been studied in the context of single-threaded programs [Chilimbi et al. 1999; Rao and Ross 2000],

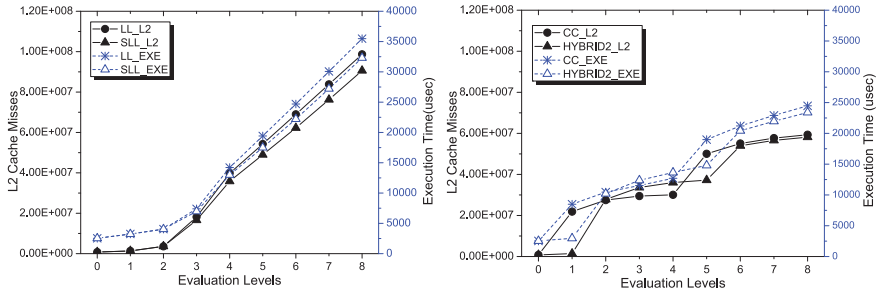


Fig. 4. Comparison of last-level (L2) cache misses and execution time for LL and SLL layouts (a) and CC and hybrid layouts (b).

with the main outcome being a cache-conscious (CC) data layout. In a CC layout, we partition a single tree into blocks according to the size of L1 or L2 cache line. For example, in Figure 3(b), if the cache line size is 16 bytes, and each node takes 4 bytes, we can put three nodes together into a single block (e.g., the dotted triangles). In each block, when the parent node is fetched into the cache, a cache hit occurs whether the left or the right child is accessed next, leading to spatial reuse.

However, for multithreaded or SIMD memory accesses, traditional CC layout does not work well, as it completely ignores interthread spatial reuse. If we have multiple trees organized with the CC layout, the loading distance for each level is still the entire tree. In our work, we slightly modify the traditional CC layout by organizing all root nodes next to each other by an LL layout, since it is obvious to improve the memory performance on SIMD architecture. We also apply the SLL strategy to organize the CC blocks into separated left and right groups. We show our CC layout in Figure 3(a) and use this version in our experiments.

4.3. Hybrid Layout

From our previous discussion, we can observe that while processing of the earlier levels of the tree gives opportunity for significant interthread spatial reuse, the loading distance increases beyond the size of a cache line after a certain level, and only intrathread locality can be exploited. Based on this observation, we design a hybrid layout schema to combine the benefit of interthread data locality of SLL for top levels and the benefit of intrathread data locality of CC layout for deeper levels. The hybrid layout is parameterized by a switching parameter, which denotes the level where we shift from an SLL layout to a CC layout.

To further validate our reasoning about relative performance of different layouts and the motivation behind the hybrid layout, we conducted several experiments. We use the B+ tree application on the Fermi architecture as an example to examine the cache behavior of different layouts. The last-level (L2) cache misses and real execution times of different layouts are shown in Figure 4. Profiling data is collected using CUDA Visual Profiler 4.1. Solid lines show the L2 cache misses, and dotted lines show the actual execution time of various versions. By comparing Figure 4(a) and Figure 4(b), we can see that for top levels, the last-level cache misses of LL and SLL layouts are lower than those with the CC layout. However, they increase rapidly, and starting from a certain level, the CC layout outperforms LL and SLL layouts. Further, we show the L2 cache misses and execution time of the hybrid layout in Figure 4(b). We can see that the overall performance of the hybrid layout is better than both SLL and CC layouts.

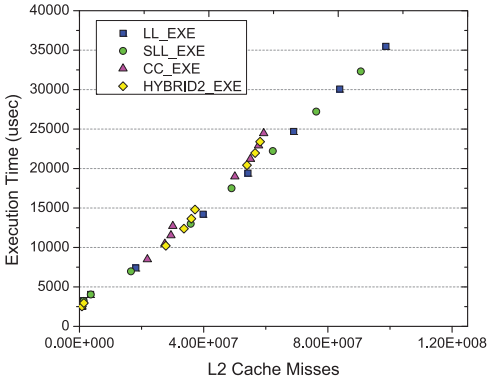


Fig. 5. Correlation between last-level (L2) cache misses and execution time (different layouts).

5. CACHE ANALYSIS MODEL FOR AUTOMATIC SELECTION OF LAYOUT

For a given application and architecture, selecting the best layout from among the ones we introduced in the previous section is a challenging problem, yet automating the selection is critical for performance portability. For example, for a given application, choosing the balance between inter- and intrathread locality is a very difficult task for the application developer, even for one architecture. For a programmer (or just a user) to achieve this balance while porting the application to another architecture is even more daunting. Application parameters like the number of bytes needed for one node of the tree, possible bias in traversing one child of the tree over others, and whether all trees are accessed with equal probability or not can all impact the choice of the layout to use. Similarly, architectural characteristics, like the size of a cache line, cache miss penalties, and degree of SIMD parallelism can impact how one layout may result in better performance over another. Driven by this motivation, we have developed a model that aids programmers in selecting the best memory layout for their application on a particular SIMD architecture.

The model is designed to be used in the following fashion. The programmer is expected to develop an application using a generic interface, where the details of the data layout are abstracted. All data layout implementations are available in a library suite. Once the optimal layout (and/or the parameters, like the switching level) are chosen by the model, the right layout is used by the application.

In general, modeling computer systems and predicting performance of a given application on a given architecture is very hard. However, by focusing on a restricted domain, we simplify the problem. For our target class of applications, the number of L2 (and L1) cache misses is an effective predictor of execution time. Figure 5 shows the relationship between L2 cache misses and execution times with the use of different layouts. We can see a very high correlation even across different layouts. More detailed profiling data (not included here) further shows that L1 cache misses can also play an important role. Based on these observations, our model captures L1 and L2 cache misses.

To understand the key insight behind our model, suppose that the tree nodes are organized by the LL layout. As we traverse through a number of trees, for the root (zero-th) level, one load from main memory is sufficient to bring in many trees (as many as fit in a cache line). Thus, the loading efficiency is 1. For the first level of the tree, we will use only 50% of the elements we have loaded (i.e., the loading efficiency is 1/2). Similarly, for the second level, the efficiency is 1/4. If only four tree nodes can be stored in one cache line, the loading efficiency remains 1/4 for subsequent levels.

Table IV. Model Parameters

Parameter	Type	Explanation	Value
N	App	No. of levels of each tree.	B+ tree $N = 9$
B	App	No. of trees to evaluate.	32
T_1	Arch	L1 cache access latency.	Assume 1 clock cycle.
T_2	Arch	L2 cache access latency (L1 cache miss penalty).	Assume $T_2 = 4T_1$.
T_M	Arch	Memory access latency (L2 cache miss penalty).	Assume $T_M = 8T_1$.
L	Both	L1 and L2 cache line size, in no. of tree nodes.	*Node_size* = 8 bytes, GPU: L = 128/8 = 16, SSE: L = 64/8 = 8
G	Both	In CC layout, the no. of levels of nodes can be held by one cache block: $G = \log_2 L$.	GPU: $G = \log_2 16 = 4$, SSE: $G = \log_2 8 = 3$
x	User	*Switch Level*, i.e., start using CC from it.	

However, a different choice of layout or any bias (e.g., to left or right children) can complicate the calculations of cache misses.

5.1. Parameters and Assumptions

Parameters. The parameters used in this model are explained in Table IV. The GPU here implies Fermi, the Tesla 10-series architecture, which does not have L1 or L2 cache, will be explained separately later. L is the cache line size in terms of the number of tree nodes, and we load L nodes into L1/L2 cache in each access. G is a parameter used for the CC layout. Recall that in the CC layout, we partition a single tree into blocks with L nodes per block in a triangular fashion. G, which is also $\log_2 L$, indicates the number of levels of nodes that fit in each block. For example, if $L = 16$, we group $G = \log_2(16) = 4$ levels of nodes into a block. The parameter G has another significance for the LL layout. Considering root as the level 0, G is the level at which the loading efficiency decreases to $1/L$ and cannot reduce any further.

Assumptions. All data is in main memory initially and both L1 and L2 cache are empty. The tree data is too big to fit into either L1 or L2 cache, or even a combination of shared memory (on GPUs) and the cache. There is no temporal reuse while executing application once. We also assume that the tree is perfectly balanced. Furthermore, the detailed calculations assume a binary tree, although only a trivial modification is needed to capture a general k-ary tree.

Initially, we focus on the hybrid layout—that is, we use either the LL or SLL layouts to exploit interthread data locality for upper levels of the tree, and use CC layout to explore intrathread data locality for the lower levels, switching at level x. Assuming that there is no bias in accessing left/right child and all trees are accessed with the same probability, our objective will be to find the proper switch level, x.

5.2. Basic Model for Balanced Accesses

Our first observation is that while using the hybrid layout with LL at initial levels, the switch to the CC layout must be made latest by the level G, because there will not be any spatial reuse with LL layout level G onward (i.e., only one node from a cache line will be read).

The memory access times for processing B trees, as a function of the switch level x, is

$$T(x) = (T_1 + T_2 + T_M) \times \frac{B}{L} \times (2^x - 1) + \left[T_1 \times (N - x) + \left\lceil \frac{(N - x)}{G} \right\rceil \times (T_2 + T_M) \right] \times B. \tag{1}$$

The two preceding terms capture the memory access times for up to the level $x-1$ and levels x through N, respectively. For the part of the trees that are organized using the LL layout, we have a total of 2^x-1 nodes for each tree, or $B\times(2^x-1)$ nodes for the B trees we are processing, which correspond to $B/L\times(2^x-1)$ cache lines. While using the CC layout, recall that G levels fit in one cache line, which means that there is a G times reuse of a single cache line. Thus, the number of cache misses for processing the last $N-x$ levels is $\lceil\frac{(N-x)}{G}\rceil$.

To get the value of x where this term is minimized, we take a derivative, resulting in

$$T'(x)=(T_1+T_2+T_M)\times\frac{2^x\ln 2\times B}{L}-\left[T_1+\frac{1}{G}\times(T_2+T_M)\right]\times B. \tag{2}$$

For $T'(x)$ in Equation (2) to be zero, we need

$$x=\log_2\frac{L\left(T_1+\frac{1}{G}\times(T_2+T_M)\right)}{\ln 2\times\ (T_1+T_2+T_M)}. \tag{3}$$

Using SLL for upper levels. We now show how the previous analysis can be extended to the SLL layout. Recall that in an SLL layout, we store the left and right parts of the trees separately. Thus, depending on the traversal, it may be possible to better exploit interthread data locality for more levels. Our model formally captures this through a deferring factor, which we empirically determine in an architecture-independent fashion. Using λ to denote such an empirically determined deferring factor, we modify the execution time expression for LL as follows:

$$\begin{aligned}T(x)\ =\ &(T_1+T_2+T_M)\times\frac{B}{L}\times(2^{x-\lambda}-1+\lambda)\\&+\left[T_1\times(N-x)+\left\lceil\frac{(N-x)}{G}\right\rceil\times(T_2+T_M)\right]\times B.\end{aligned} \tag{4}$$

Similarly, by taking the derivative of T in terms of x, and checking when it becomes zero, we have

$$x=\log_2\frac{L\left(T_1+\frac{1}{G}\times(T_2+T_M)\right)}{\ln 2\times(T_1+T_2+T_M)}+\lambda. \tag{5}$$

Equation (5) shows that λ also affects the split level. By analysis of performance with datasets that have bias, we have found that $\lambda=1$ is quite effective, so we use this value in all of our experiments.

5.3. Capturing Biased Accesses

In many cases, probability of accessing different children of any node is different. Without loss of generality, we assume that the child more likely to be accessed is organized to the left.

We introduce a bias metric β to indicate the percentage of inputs falling into the left-most path of all trees. Then, if $T1(x)$ (defined shortly) is the time spent when an input involves all left-most paths and if $T2(x)$ is the time spent for all other inputs, we have

$$T(x)=T2(x)\times(1-\beta)+T1(x)\times\beta. \tag{6}$$

$T2$ can be calculated using Equation (4). For calculating $T1$, we perform the following analysis. If all queries fall into the left-most path of each tree, the SLL layout is

preferred and then the loading time of a bag of trees is

$$T(x) = (T_1 + T_2 + T_M) \times \frac{B}{L} \times (1 + 2(x-1)) + \left[T_1 \times (N-x) + \left\lceil \frac{(N-x)}{G} \right\rceil \times (T_2 + T_M) \right] \times B. \quad (7)$$

An analysis of the preceding expression shows that when $x = N$, $T1(x)$ is the minimum. We choose x to find the minimum value, where the weighted sum of $T1(x)$ and $T2(x)$ is minimized. Intuitively, we can see that if there is more bias, we can better exploit interthread locality. Thus, either an SLL layout or a hybrid layout with SLL at the top several levels is optimal.

β is an architecture independent parameter, which depends on the application, and more particularly, the dataset. To obtain the value of β, we employed the following strategy. First, we randomly sample 5% of the input data. We process these inputs and calculate the probability of reaching each leaf node (denoted as P_i for the leaf node i). We then estimate β as the sum of the probability of reaching the K left-most leaf nodes—that is, $\beta = \sum_{i=1}^{K} P_i$—in which $K = Num_{total\_leaf} \times 5\%$.

5.4. Impact of Sparse Buckets Accesses

As we discussed previously, in an application like B+ tree forest, each tree can have an imbalanced number of inputs in that tree's bucket. This situation, which we refer to as sparse buckets accesses, can change the relative performance if we use different layouts.

Let us revisit Equation (1) and see how to modify it to handle this particular situation. If the probability of accessing any particular tree is high, each of the blocks at the upper level still need to be loaded into memory. However, at the lower levels, when we are using the CC layout, not all blocks will be loaded. Thus, we can simply apply a scaling factor θ, $\theta < 1$, to the second part of Equation (1). Analyzing when this expression is minimized, we now get the value of x as $\log_2 \frac{L(T_1 + \frac{1}{G} \times (T_2 + T_M)) \times \theta}{\ln 2 \times (T_1 + T_2 + T_M)}$. Thus, it is preferable to focus on intrathread spatial reuse starting with even earlier levels.

To summarize, θ is a dataset-dependent parameter, which we can estimate by randomly sampling from the said dataset.

5.5. Modeling a System without L1/L2 Cache

Our discussion so far has assumed presence of an L1 and L2 cache. An architecture like that of the Tesla 10-series GPU does not have L1 or L2 cache, although it does have support for coalesced accesses—that is, simultaneous access to consecutive memory locations by different threads are faster than random accesses.

Our model can also be applied to such an architecture with small modifications. Particularly, we set $G = 1$, and L is used to show the coalesced access block size. By applying similar calculations to Equations (1) and (2), we get $T'(x) = 0$ when $x > \log_2 L$ ($\log_2 L$ is the last level where it is possible to utilize coalesced accesses for LL layout). This implies that we can only utilize interthread data locality for such an architecture, and our hybrid layout does not provide any further benefits for deeper levels. Furthermore, with the SLL layout, it is possible to further exploit the interthread data locality for a few additional levels.

6. EXPERIMENTAL RESULTS

In this section, we describe our experimental results. We had the following goals in our experiments: (1) examining the speedups obtained using SIMD parallelism for our

Table V. Characteristics of Datasets Used in Our Experiments

Dataset	#Tree	#Ave_Node	Path_Leng	Ave_Path_Leng	Bias
B+ tree	3,584	513	8–11	9.0	various
Poker	3,584	249	4–10	7.3	0.51
Shuttle	3,584	217	4–10	7.5	0.55
Abalone	3,584	333	5–12	8.0	0.52
Satellite	3,584	353	4–12	8.2	0.55
Microsoft	3,372	239	1–45	11.34	0.8

target class of applications, (2) understanding the relative performance with different layouts, and (3) validating the analytical model that we have developed.

Platforms. We conduct our experiments on the following three machines: (1) C2050 GPU with the Fermi architecture, connected to an Intel Xeon E5630 CPU (2.53GHz frequency); (2) Quadro FX 5800 GPU with the Tesla architecture, connected to Quad-Core AMD Opteron(tm) Processor 2380 (2.49GHz frequency); and (3) Intel Xeon E5420 CPU (2.5GHz frequency) with Streaming SIMD Extension 4 (SSE-4). Sequential and CUDA codes are compiled by g++ and nvcc, respectively, with O3 optimizations, whereas SSE codes are compiled with Intel ICC (Intel Parallel Composer 2011) compiler to fully utilize the SSE unit. We run all programs 30 times, and speedup numbers include the mean and 95% confidence interval of the mean. For CUDA versions, the execution time difference across different runs is very small, and therefore we omit the error bars.

Methods. For the random forest application, we used trees and datasets from two different sources. The first is a popular open source numerical analysis and data processing library, ALGLIB, with which we used four datasets distributed by the UCI Machine Learning Repository—Poker, Shuttle, Abalone, and Satellite. The second is an internal random forest created for a large Microsoft product that has its own associated datasets. For the B+ tree application, our evaluation was based on the experiments reported in the literature [Wu et al. 2010]. Based on this study, we establish a tree forest with different degrees of left bias input datasets: unbiased (50% bias), 62.5% bias, 75% bias, and 87.5% bias. Table V summarizes the basic information of tree forests used for both our B+ tree and random forest applications. For regular expression, we search the King James Bible for up to 10 different regular expressions. Each regular expression follows the pattern $.*ab$, where the characters a and b are unique for each regular expression. To match N regular expressions, we combine them using the choice operator.

For both Tesla 10 and Fermi architectures, we used the available shared memory as a buffer to hold input features and evaluation task queues. For the Fermi architecture, we used 48KB shared memory (and thus, a 16KB L1 cache) to hold both the evaluation buffer and also some of the top-level nodes of the trees.

Baselines. In both random forest and B+ tree applications, we use a sequential, pointer-based CPU implementation as a baseline. We wrote our own sequential B+-tree baseline, and in the random forest application, we use either the ALGLIB original sequential implementation or the Microsoft sequential implementation. To focus on speedups obtained using SIMD parallelism, we also created locality optimized sequential versions. This version uses a linearized layout, with a DF traversal, and uses *bagging* or *tiling* to organize B trees together into a single bag. This version is referred as *DF_Seq* in our description. The value of B that leads to the best performance was empirically determined and used in our sequential version. For regular expression application, we compare our approach to GNU `grep`, which is chosen for two reasons. First, like our regular expression engine, it counts matches and matches regular

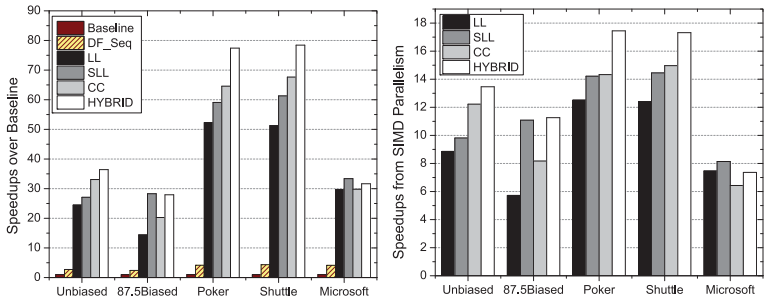


Fig. 6. B+ tree (two datasets) and random forest (three datasets) on Fermi GPU. (a) Speedups of different versions over sequential baselines. (b) Speedups from SIMD parallelization (same layout used for sequential execution, i.e., LL shows the speedup of SIMD parallel with LL over sequential with LL, and so on).

expressions from the POSIX Extended Regular Expression syntax. Second, GNU `grep` is known to be fast.[3]

6.1. Speedups and Performance with Different Layouts

In this section, we demonstrate the efficacy of our different layouts. In particular, we demonstrate that our layouts are able to (1) significantly increase the performance of these irregular applications and (2) enable the use of SIMD architecture for this class of application.

6.1.1. B+ Tree. We evaluated the B+ tree application on both the Fermi GPU and the SSE architecture. Results from two datasets, the unbiased traversal and the 87.5%, are reported here. In Figure 6(a), we show the speedups of different versions over the sequential baseline on Fermi GPU architecture. We see that using SIMD execution, we gain around 25X to 36X speedup over the sequential baseline for the unbiased traversal, and 15X to 29X speedup for the biased traversal. To separate the speedup from GPU's parallelism from linearization of the structures, in Figure 6(b) we show the speedup of CUDA versions over the sequential version with the same layout. For unbiased traversal, the Fermi GPU architecture enables 9X to 13X speedup, whereas for biased traversal, 6X to 11X speedups are seen. Considering the irregular nature of these applications, we consider these speedups to be substantial, and the use of GPUs toward accelerating this application is justified.

From Figure 6(a), we also see that the SLL and hybrid layouts are better than LL and CC. For unbiased traversal, the hybrid layout shows the best performance, since it allows benefits from interthread spatial reuse at earlier levels, and intrathread spatial reuse at later levels. For biased traversal, SLL layout allows benefits from interthread reuse for deeper levels as well and has the best speedups. It should all be noted that for the 87.5% bias case, the switch level used is not optimized for bias, which cause the hybrid layout to perform worse. Overall, up to a factor 2X difference in performance can be seen from the choice of layout from among the four layouts that we have developed.

The same experiments were repeated on the SSE architecture, and the results are shown in Figure 7. Similar trends can be seen, and the only difference is that the

[3]In a recent post to the freebsd mailing list, entitled "Why GNU grep Is Fast," the author of GNU `grep` describes why his implementation is fast; GNU `grep` uses the Boyer-Moore [Moore and Boyer 1977] algorithm for sublinear search. It also uses a DFA-based graph traversal once it finds a position in the input string to match against text.

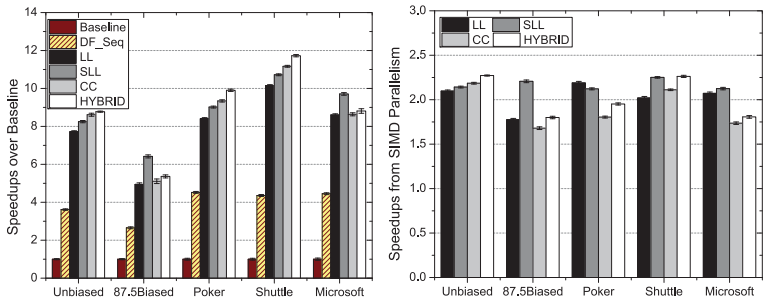


Fig. 7. B+ tree (two datasets) and random forest (three datasets) on SSE. (a) Speedups of different versions over sequential baselines. (b) Speedups from SIMD parallelization (same layout used for sequential execution, i.e., LL shows the speedup of SIMD parallel with LL over sequential with LL, and so on).

optimization of memory layouts brings relatively smaller benefits on SSE than on the Fermi GPU. There are two reasons for this: (1) the SIMD lane width of SSE is much narrower (i.e., only four-way parallelism is possible), and thus the data requirements in each cycle are modest, and (2) the modern CPU memory hierarchy is more advanced than GPU, with better prefetching as well as prediction strategies, which makes it possible to avoid some of the cache misses.

6.1.2. Random Forest. We also conducted similar experiments on the random forests of ALGLIB and Microsoft. Due to the space limitation, we only show the results from two datasets for ALGLIB random forest: Poker and Shuttle, as the trends from Abalone and Satellite almost exactly match the trends for these two.

In Figure 6(a) and Figure 7(a), we show the speedups of different versions over the baseline on GPU and SSE, whereas in Figure 6(b) and Figure 7(b), we show the speedup of GPU and SSE versions over corresponding best sequential versions. The speedups are very similar to those we obtained from B+ tree. One notable difference is that the random forest of Microsoft shows only 8X speedup from the GPU. This is because in the dataset used, the input feature vector is very large (2,648 float numbers), and shared memory cannot be used in the same fashion as in the UCI datasets. Among different layouts, the best performance is obtained from the hybrid layout for Poker and Shuttle, and SLL for Microsoft. Considering that Poker and Shuttle is almost unbiased and Microsoft has a significant bias, these results are consistent with what we saw with the two datasets of B+ tree.

6.1.3. Regular Expression Matching. We now investigate the performance of our SIMD interpreter on regular expression matching. For this application, we use a simple LL layout because the graphs generated by our regular expressions are small and fit easily in L1, so memory optimizations are not as important as in the random forest application. We also only report the experiment results for SSE architecture for this application, because in recent years, there have been many application-specific works that have heavily optimized the GPU's implementation of this application [Cascarano et al. 2010; Zu et al. 2012]. Note that because we can pack instructions into a `byte`, our SIMD interpreter can traverse up to 16 graphs in parallel for this application.

Figure 8 shows the speedup of our approach. A bar on this graph (x, y) gives the speedup over GNU `grep` (y), varying the number of regular expressions, or fine-grained tasks, executed. GNU `grep` at 1.0 is the baseline. It is very fast for the first two regular expressions, as it uses Boyer-Moore to perform a sublinear search over the input string. However, after three or more regular expressions, GNU `grep` cannot use Boyer-Moore,

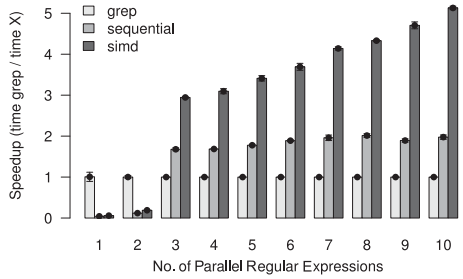


Fig. 8. Speedup of the SIMD interpreter over GNU `grep` regular expressions.

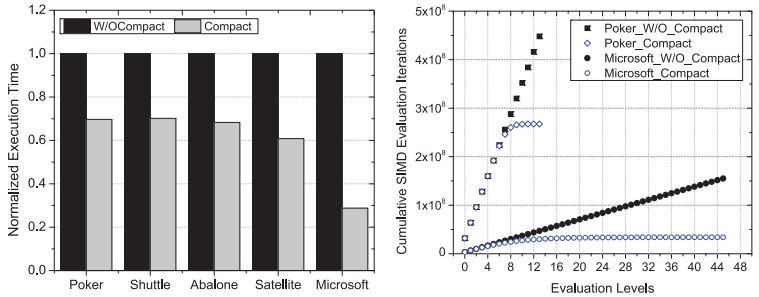


Fig. 9. Stream compaction. (a) Speedup improvements—random forest using five datasets. (b) Reduction in workload—Poker and Microsoft datasets.

as the resulting regular expression gets too complicated. After three parallel regular expressions, the sequential interpreter is 1.7X faster than GNU `grep`. This is due to the regular LL access pattern of our interpreter. However, the speedup for the SIMD interpreter linearly increases as we add fine-grained tasks. The SIMD interpreter is anywhere from 3X to 5X faster when searching for three or more parallel regular expressions.

6.1.4. Improvements from Stream Compaction for SSE Implementation. In this part, using one of the applications (random forest), we quantify the gains from our light-weight *stream compaction* optimization on SSE architecture.

Figure 9(a) shows the comparison of execution times among the SIMD code with and without stream compaction for each of the five datasets. The results show that for datasets with a smaller variation in path lengths, such as Poker and Shuttle, the stream compaction method gives around 30% speedup over the unoptimized version. For the dataset that has a larger variation in path lengths (i.e., the Microsoft dataset), stream compaction gives more than 70% speedup.

To further study the reasons for these speedups, in Figure 9(b) we show the workload reduction by the stream compaction method using two representative datasets: Poker and Microsoft. Specifically, Poker represents the case with a smaller variation in path lengths, whereas Microsoft involves a much larger variation in path lengths. The x-axis here is the evaluation level, and the y-axis is the cumulative number of SIMD evaluation iterations (i.e., the workload on the SIMD lanes). We can see that for the Poker dataset, our stream compaction method is able to reduce around 40% of the workload, with most gains seen from levels 8 through 12. For the Microsoft dataset, the benefits are seen even at earlier levels of the tree and, overall, add up to 80% of the number of iterations needed. By comparing the workload reduction (80% and 40%)

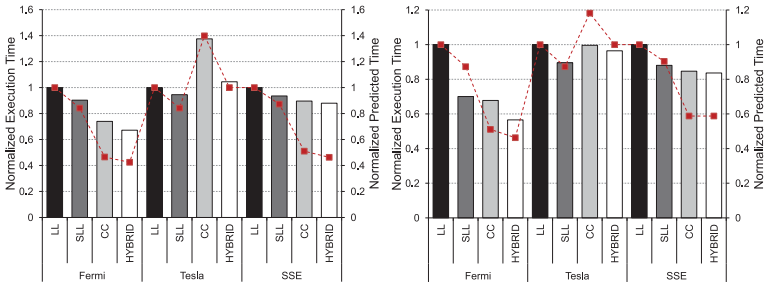


Fig. 10. Real and predicted execution times with different layouts and architectures. (a) B+ tree forest with unbiased traversal. (b) Random forest with satellite dataset.

and the execution time reduction (70% and 30%), we see that stream compaction only introduces a 10% overhead.

6.2. Model Validation

Our model is accurate and thus enables performance portability for this class of application on SIMD hardware. In this section, we demonstrate our model's accuracy on a wide variety of SIMD architectures. Our model, as described in the previous section, only calculates the memory access times. For a more direct comparison with real execution times, the predicted values were normalized to execution times by using linear regression with a small number of sample values.

6.2.1. Choosing Layouts on Different Architectures. Our analytical model is designed to support automatic optimization, with the goal of performance portability across different architectures. Thus, to evaluate its effectiveness, we first examine its ability to choose best layout for two different applications on three different architectures. In Figure 10, we compare observed and predicted execution time with four different layouts on two GPU architectures: Fermi (Tesla C2050) (with L1/L2 caches) and Tesla (Quadro FX5800) (without caches), and the SSE architecture with a sophisticated cache hierarchy. The left and right y-axes correspond to measured and model predicted execution times. The solid bars report measured performance, whereas the dotted line shows the model predicted times.

Figure 10(a) reports results from the B+ tree application, using a dataset where there is no bias. We can see that for all three architectures, our model is able to predict the layout that will result in the lowest execution time. Moreover, we can even predict the relative execution times for the four layouts on all architectures. Particularly noteworthy is that the relative performance trends are very different for Fermi and Tesla, yet they can be captured by our model. On Fermi architecture, the hybrid layout results in the best performance, around 35%, 25%, and 10% faster than LL, SLL, and CC, respectively. On the Tesla architecture without cache hierarchy, the SLL layout shows the best performance, whereas CC shows the worst, since we cannot exploit any intrathread locality here. The trends on SSE are quite similar to those on Fermi, although the relative differences between different layouts are much smaller. This is because of support for aggressive prefetching and limited degree of parallelism, both of which reduce the performance impact due to spatial reuse.

We can see that our model does not always predict the precise execution time. This is because we are using a simple model, which only captures memory access times. Factors related to degree of parallelism in the application are not captured. However,

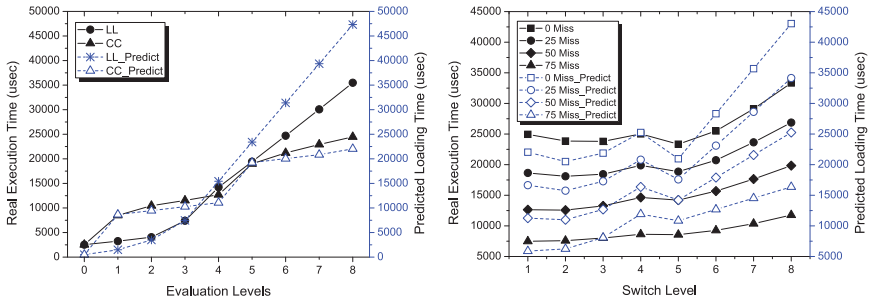


Fig. 11. Comparing real execution and model predicted times. (a) For each level of the tree: LL and CC layouts, B+ tree on Fermi. (b) For different degrees of Sparse Accesses.

our simple model is able to achieve our goal of predicting the relative performance with different layouts.

We repeat the same experiment on the random forest application, using the ALGLIB library tree with the Satellite dataset from the UCI repository. The results are shown in Figure 10(a), which present similar trends. Again, our model is able to predict which layout will result in the best performance in each case.

To further examine the efficacy of the analytical model, we carefully studied how it predicts the evaluation time for different levels of trees. The results are shown in Figure 11(a). The x-axis is the evaluation level of the tree, and the left and right y-axes are the measured and the predicted times, respectively. Note that in all double-y-axis figures of this section, the solid lines correspond to the left y-axis, and the dotted lines correspond to the right y-axis. We have compared two contrasting layouts: LL and CC. For LL, both the measured and the model predicted times show an exponential increased, followed by a linear behavior. The model predicted times follow the shape of the curve of the measured times even though there are differences in the absolute values. Again, with the CC layout, the predicted execution time curve matches the shape of the curve of the measured execution times, and both show stage-increasing behavior. Similarly, the level at which the CC starts outperforming LL (level 4) can be correctly predicted by our model.

6.2.2. Handling Application Characteristics. Besides performance portability across different architectures, another goal of our analytical model is to be able to choose the appropriate layout for applications that have different characteristics. We now show how the model is able to predict performance when there can be bias in traversal or sparse accesses.

An important factor that impacts the relative performance of different layouts is the bias degree of the traversal in each tree. Particularly, the SLL layout can be more effective when there is a bias, and similarly, in using the hybrid layout, it helps to switch at a deeper level when there is a bias. We now examine how our prediction model can help choose the appropriate layout, reporting the results in Figure 12, looking at an unbiased case and three different levels of bias. We consider the hybrid layout and vary the switch level—that is, the level at which we start using the CC layout. Again, we can see that the shape of curves for the real and predicted times match well. The performance obtained at the switch level predicted by the model is either the best, or very close to the best, performance observed experimentally.

Another important application factor is the sparsity level—that is, the probability that any given tree will not be accessed during one execution. We have again compared the observed and predicted times. In Figure 11(b), we show the cases with sparsity

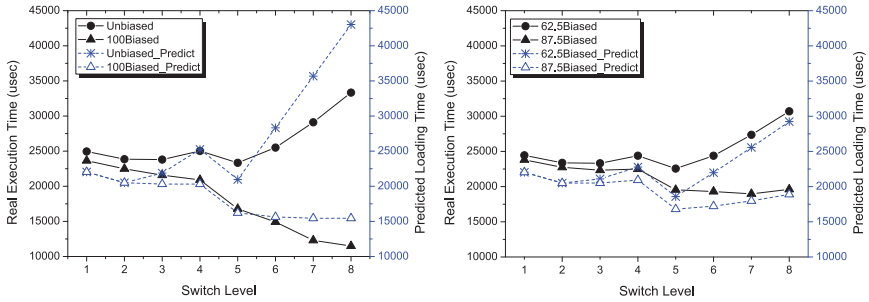


Fig. 12. Comparing real and predicted execution times with different bias levels: B+ tree on Fermi GPU.

levels varying from 0% to 75%. Again, we use the hybrid layout and vary the switch level. By comparing the measured and predicted times, we see that the corresponding curves match very well, and the switch level leading to the best performance can be correctly predicted.

The parameters and factors that we consider in this section include ones that are specific to the architecture (and independent of the application) and those that depend on the application (but independent of the architecture). The latter parameters are easy to obtain from the architecture's description. In contrast, application parameters can be more difficult to obtain. Application-specific parameters include (1) whether or not there is bias, or whether or not all trees are equally likely to be accessed, and (2) specific levels of bias or sparsity in accesses. The former is readily available to application developers since it depends on the underlying algorithm. Sampling and/or offline profiling could be used for the latter, although it may still be inaccurate. From the results presented in this section, we can observe that modest changes in bias or sparsity do not impact our model's choice of the layout, and thus some inaccuracies in obtaining these parameters are tolerable. For example, in Figure 12(b), for both 62.5% and 87.5% bias, the best performance is obtained with 5 as the switch level. Moreover, our framework still allows large speedups over the baseline even if a nonoptimal switch level is chosen.

7. RELATED WORK

This section compares our work with related research efforts from other groups, especially the work of parallelizing irregular data structures on various SIMD architectures and the efforts of increasing irregular data locality in both single-core and multicore environments.

7.1. Execution of Irregular Data Structure Traversal Algorithms on Parallel Architectures

Earlier work had used very sophisticated compiler analysis to automatically determine parallelism in pointer-based programs [Ghiya et al. 1998]. More recently, the Galois project has extensively considered parallelization of irregular applications [Kulkarni et al. 2009; Méndez-Lojo et al. 2010]. Their focus is coarse-grained or MIMD parallelism, whereas our focus is SIMD execution.

There are also many efforts focusing on manual optimization of this class of applications on SIMD and vector units. Key recent efforts include the work by Sewall et al. [2011] and Kim et al. [2010]. This work considers simultaneously processing multiple inputs on a single data structure. We are focusing on processing one input point across multiple pointer-based data structures and focus on a more general interpretation system.

Execution of tree and graph traversal algorithms on SIMD (mostly GPU) has been a popular topic in recent years. As GPGPUs were emerging, Harish and Narayanan [2007] designed a set of algorithms to map graph algorithms to the GPU architecture. More recently, many others have worked on this problem with many efforts focusing on BF traversals [Luo et al. 2010; Agarwal et al. 2010; Hong et al. 2011; Merrill et al. 2012], which is a key kernel in many applications, and others focusing on single source shortest path or other interesting graph algorithms [Delling et al. 2011; Solomon et al. 2010]. Our work is distinct in several ways. First, we consider traversals over a collection of trees, which leads to a different set of challenges for memory locality. Second, our focus is on performance portability, which has not been the topic of prior studies.

More closely related to the class of applications we study in our work, tree forest applications like decision trees and suffix trees have also been studied on GPUs [Sharp 2008; Schatz et al. 2007; Trapnell and Schatz 2009]. Especially, Sharp [2008] has parallelized decision tree and forest traversal on GPUs. The work is based on using a GPU's texture memory and does not apply to the SSE units we have considered. Moreover, none of them have carefully studied the effect of different memory organizations architecture—that is, they usually implement one specific layout. In addition, this prior work does not use any form of analytical modeling to achieve performance portability.

Similarly, regular expression traversal has been implemented on GPUs [Vasiliadis et al. 2009] and Cell processor [Scarpazza and Russell 2009]. Cascarano et al. [2010] also designed an NFA-based regular expression engine focusing on the GPU's architecture, which has been further improved by Zu et al. [2012]. Our work is distinct in considering SSE parallelism and locality issues related to modern uniprocessors. Prior to the interest in SIMD or many-core execution, many efforts focused on vectorization of pointer-based applications. Lars and Hernquist [1990] and Junichiro and Makino [1990] vectorized tree traversals but considered only a single tree.

More recently, Burtscher et al. [2012] conducted a quantitative study of various irregular programs on GPUs in which they define two measures of irregularity at the warp level called *control-flow irregularity* and *memory-access irregularity*, and discuss the effect and trade-off between them on kernel performance. The focus of our work is on designing a virtual machine technique to address challenges of irregular applications.

Another class of irregular applications involve sparse matrices and/or indirection arrays. SIMD and GPU parallelization and optimization of these applications has been studied in recent years. For example, Kim and Han [2012] proposed a code generation method to vectorize indirection of array-based loops, and Zhang et al. [2011] designed a set of strategies to optimize such irregular applications on the GPU's architecture.

Processing of MIMD tasks on SIMD machines has received considerable attention in the past. For example, Hanxleden and Kennedy [1992] developed loop transformation techniques (focused on array-based programs) to achieve this goal. Prins and Palmer [1993] had a similar focus but targeted vectorization. Dietz and Cohen [1992] described a more general scheme. Blelloch et al. [1994] and Hardwick [1996] focused on exploiting nested data parallelism, similar in spirit to our use of data parallelism to handle irregular applications. Our work has considered specific challenges arising for pointer-based traversals, which have not been considered in the past. We have also developed optimizations that are critical for performance on today's processors (e.g., locality, as more applications have become memory bound over time).

7.2. Improving Data Locality of Irregular Data Structures

Improving memory locality of irregular data structures has also been studied in the past. Chilimbi et al. [1999] developed a set of CC structures, and Rao and Ross [2000] proposed a CC B-tree structure. This early work considered sequential (single thread)

execution and forms the basis for the one of the layouts we will consider while optimizing for SIMD execution. Kim et al. [2010] designed an architecture-sensitive binary search tree for both CPUs and GPUs. They did not consider the processing of multiple trees concurrently and thus faced a distinct set of data locality issues.

In a recent effort, Jo and Kulkarni [2011, 2012] designed a tiling-like transformation to improve the performance of irregular data structure traversal, and more recently, Jo et al. [2013] applied this dynamic reordering transformation as a scheduling for SIMD execution and achieved good performance. Their primary focus was on temporal reuse, which is complementary to our work on spatial locality.

7.3. Exploring Interthread Data Locality in Multithreaded Environment and Cache Modeling

There have also been many efforts on exploiting interthread data locality in multithreaded environments. In this area, Meng et al. [2010] designed a symbiotic affinity scheduling (SAS) algorithm to maximize the cache locality of the threads on the same core. Che et al. [2011] proposed an API, Dymaxion, to help programmers reorganize the data to achieve better data locality in a heterogeneous environment. Zhang et al. [2010] proposed a method to transform programs in a cache-sharing-aware manner to improve the performance, whereas Jang et al. [2011] provided a set of techniques to transform the data according to different memory access patterns to improve the performance on both AMD and NVIDIA GPUs. Considering performance fairness in multiprocessor environments, Zhou et al. [2009] designed a mechanism to share the cache among concurrent applications. Ding et al. [2011] proposed a runtime library, *User Level Cache Control*, for programmers to explicitly manage and optimize the last-level cache for datasets in multithreaded programs. More recently, Unkule et al. [2012] presented a software framework to analyze and restructure the GPU kernels to explore interthread data locality. The distinct aspects of our work are as follows: (1) we are focusing on improving inter-thread spatial reuse across concurrent threads from the same application, whereas most of the preceding work considers capacity and conflict misses from different applications, and (2) our focus is on detailed analytical modeling of one application with the goal of performance portability.

Cache behavior modeling and analysis is widely used as part of restructuring compilers that focus on scientific (array-based) programs. Earlier work in this area includes that of Porterfield [1989] and McKinley [1998]. More recently, Cascaval and Padua [2003] proposed a machine independent model to estimate cache misses during compile time based on stack algorithms. Zhong et al. [2004] developed a model based on *Whole Program Reference Affinity*. Our work is distinct in considering a different class of applications and combining intrathread reuse with interthread reuse for SIMD.

8. CONCLUSION AND FUTURE WORK

This article explains how to extract SIMD parallelism from applications that traverse irregular data structures such as trees and graphs. As SIMD execution units become more common and capable in the near future, it becomes increasingly pressing to find general techniques to exploit the power of this hardware in new and broader contexts. Our work describes one such approach, which is to traverse and compute on multiple, independent, irregular data structures in parallel using a targeted virtual machine running on various SIMD architectures. By scheduling operations from the virtual machine and implementing a number of optimizations, we have shown substantial speedups on three latency-critical applications.

Moreover, optimizing an application on any one particular architecture is a challenging task—optimizing that application for several architectures is a daunting, if not impossible, task. In other words, it is difficult for programmers to guarantee an application's performance portability.

For SIMD architectures, the memory hierarchy is often the bottleneck to peak performance. In this article, we introduce data layout optimizations for a common class of memory-bound applications designed to balance intra- and interthread spatial locality. Further, to remove the burden on a programmer from deciding which data layout to choose for which SIMD architecture, we have developed an accurate model that enables performance portability for these applications, and extensively validate it across different applications and architectures. Our experiments' results show that our model is able to capture the affect factors of performance from both the architecture and application sides, which provides optimized configurations for the users' program and offers as much as around 45% speedup compared to the suboptimal solution. Although our work has been in context of a specific class of applications, the main underlying idea of analytically choosing and/or combining intra- and interthread locality is broadly applicable, especially as multicore and many-core architectures become more popular.

Our work can be extended in multiple directions. One of the challenges will be handling applications where graphs or trees may change over time. Similarly, although we can target both SSE and GPUs, coupled CPU+GPU architectures may provide new opportunities and challenges for our work. We also need to systematically evaluate the programmability of our system.

REFERENCES

V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE, 1–11.

G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. 1994. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing* 21, 1, 4–14.

L. Breiman. 2001. Random forests. *Machine Learning* 45, 1, 5–32.

M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'12)*. IEEE, 141–151.

N. Cascarano, P. Rolando, F. Risso, and R. Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5, 20–26.

C. Cascaval and D. A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*. ACM, 150–159.

S. Chatterjee, G. E. Blelloch, and M. Zagha. 1990. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (SC'90)*. 666–675.

S. Che, J. W. Sheaffer, and K. Skadron. 2011. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. IEEE, Article 13.

T. M. Chilimbi, M. D. Hill, and J. R. Larus. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference onProgramming Language Design and Implementation (PLDI'99)*. ACM, 1–12.

D. Comer. 1979. The ubiquitous B-tree. *ACM Computing Surveys* 11, 2, 121–137.

R. Cox. 2007. Regular Expression Matching Can Be Simple and Fast. Available at http://swtch.com/~rsc/regexp/regexp1.html.

D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. 2011. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE, 921–931.

H. G. Dietz and W. E. Cohen. 1992. *A Massively Parallel MIMD Implemented by SIMD Hardware?* Technical Report. Purdue University.

X. Ding, K. Wang, and X. Zhang. 2011. ULCC: A user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, 103–112.

F. Franchetti and M. Puschel. 2002. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE, 20–26.

C. Garca, R. Lario, M. Prieto, L. Piuel, and F. Tirado. 2003. Vectorization of multigrid codes using SIMD ISA extensions. *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*. IEEE, 58.1.

R. Ghiya, L. J. Hendren, and Y. Zhu. 1998. Detecting parallelism in C programs with recursive data structures. In *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*. Springer, 159–173.

J. Han, H. Cheng, D. Xin, and X. Yan. 2007. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery* 15, 1, 55–86.

R. von Hanxleden and K. Kennedy. 1992. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, 188–199.

J. C. Hardwick. 1996. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *Proceedings of the 1st International Workshop on High-Level Programming Models and Supportive Environments (HIPS'96)*. 105–114.

P. Harish and P. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. 197–208.

M. Harris, S. Sengupta, and J. D. Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU Gems* 3, 39, 851–876.

L. Hernquist. 1990. Vectorization of tree traversals. *Journal of Computational Physics* 87, 1, 137–147.

S. Hong, T. Oguntebi, and K. Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 78–88.

N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2000. 2.44-GFLOPS 300-MHz floating-point vector-processing unit for high-performance 3D graphics computing. *IEEE Journal of Solid-State Circuits* 35, 7, 1025–1033.

B. Jang, D. Schaa, P. Mistry, and D. Kaeli. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1, 105–118.

Y. Jo, M. Goldfarb, and M. Kulkarni. 2013. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, 363–374.

Y. Jo and M. Kulkarni. 2011. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. ACM, 463–482.

Y. Jo and M. Kulkarni. 2012. Automatically enhancing locality for tree traversals with traversal splicing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, 355–374.

C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, 339–350.

S. Kim and H. Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, 55–64.

M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. 2009. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, 3–14.

L. Luo, M. Wong, and W. Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference (DAC'10)*. ACM Press, New York, NY, 52–55.

J. Makino. 1990. Vectorization of a treecode. *Journal of Computational Physics* 87, 1, 148–160.

K. S. McKinley. 1998. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 9, 8, 769–787.

M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. 2010. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, 3–14.

J. Meng, J. W. Sheaffer, and K. Skadron. 2010. Exploiting inter-thread temporal locality for chip multithreading. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. IEEE, 1–12.

D. Merrill, M. Garland, and A. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM Press, New York, NY, 117–128.

J. S. Moore and R. S. Boyer. 1977. A fast string searching algorithm. *Communications of the ACM* 20, 10, 762–772.

M. Nicola and J. John. 2003. XML parsing: A threat to database performance. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM'03)*. ACM, 175–178.

A. Porterfield. 1989. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Rice University, Department of Computer Science.

J. Prins and D. W. Palmer. 1993. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*. ACM, 119–128.

J. Rao and K. A. Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, 475–486.

B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE, 1–10.

M. Roesch. 1999. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA'99)*. USENIX Association, Berkeley, CA, 229–238.

T. Rognes and E. Seeberg. 2000. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 16, 8, 699–706.

D. P. Scarpazza and G. F. Russell. 2009. High-performance regular expression scanning on the Cell/B.E. processor. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM Press, New York, NY, 14–25.

M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. 2007. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics* 8, 1, 474.

J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11, 795–806.

T. Sharp. 2008. Implementing decision trees and forests on a GPU. In *Computer Vision—ECCV 2008*. Lecture Notes in Computer Science, Vol. 5305, Springer, 595–608.

S. Solomon, P. Thulasiraman, and R. K. Thulasiram. 2010. Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *Proceedings of the IEEE 12th International Conference on High Performance Computing and Communications (HPCC'10)*. IEEE Computer Society, Washington, DC, 297–304.

K. Thompson. 1968. Programming techniques: Regular expression search algorithm. *Communications of the ACM* 11, 6, 419–422.

C. Trapnell and M. C. Schatz. 2009. Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing* 35, 8, 429–440.

S. Unkule, C. Shaltz, and A. Qasem. 2012. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer-Verlag, Berlin, 21–40.

G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis. 2009. Regular expression matching on graphics hardware for intrusion detection. In *Recent Advances in Intrusion Detection*. Lecture Notes in Computer Science, Vol. 5758. Springer, Berlin, 265–283.

S. Wu, D. Jiang, B. C. Ooi, and K. Wu. 2010. Efficient B-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment* 3, 1, 1207–1218.

X. Yan and J. Han. 2002. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*. IEEE, 721–724.

E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 369–380.

E. Z. Zhang, Y. Jiang, and X. Shen. 2010. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, 203–212.

Y. Zhong, M. Orlovich, X. Shen, and C. Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, 255–266.

X. Zhou, W. Chen, and W. Zheng. 2009. Cache sharing management for performance fairness in chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE Computer Society, Washington, DC, 384–393.

Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, 129–140.

Received July 2013; revised January 2014; accepted February 2014